

Application of the Goertzel Algorithm.

Craig C. Drennan
November 26, 2002

I. Introduction.

The high sampling rates of currently available analog to digital converters make it possible to directly digitize and process the radio frequency signals generated by beam measurement instruments here at the lab. This includes the 53 MHz signals produced by BPM beam position monitors.

There are several digital signal processing methods that can be employed alone or in combination to determine signal amplitude and phase once the signals are digitized. These include digital filtering, digital down conversion and decimation, and Discrete Fourier Transform methods.

In most beam parameter measurements we are interested in the amplitude of the signals at only a single frequency. This may be the original frequency of the signal before it is digitized, or an alias of this frequency if an under-sampling approach is used.

This paper explores the Goertzel algorithm which can compute a single point in the discrete Fourier transform.

II. The Goertzel Algorithm.

There are three references given at the end of this paper that are generally available that describe the algorithm with various degrees of rigor. The most compact description is available in Kevin Banks article from "Embedded Systems Programming" magazine [1]. This article is currently available for download at

<http://www.embedded.com/story/OEG20020819S0057>

A slightly more formal presentation can be found in [2] and at

http://www.analogdevices.com/library/dspManuals/pdf/Volume1/Chapter_14.pdf

The mathematical explanation and derivation can be found in Oppenheim and Schafer [3]. This text is available in the FNAL library as call number TK5102.5 .O2452 1989.

<http://fnlib2.fnal.gov/MARION/AAA-9494>

II.1 Comparison to the DFT

The definition of an N-point Discrete Fourier Transform, or DFT, is as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{n \cdot k}$$

where $k = 0, 1, \dots, N-1$ and $W_N^{n \cdot k} = e^{-j(2\pi/N) \cdot n \cdot k}$

Each point in the output of the DFT is a complex number from which you can compute amplitude and phase of a particular frequency component of the original sequence $x(n)$. Each k is an index for a discrete frequency bin. The frequency resolution, or bin width, of the DFT output is

$$\text{BinWidth} = \frac{F_s}{N} \quad \text{where } F_s = \text{Digitizer Sampling Frequency.}$$

Note that the inverse of the bin width is the duration in seconds of our sampled signal. In many of the beam parameter measurements made, the signals of interest are available for only a short interval in time. In these situations it is desirable to digitize as much of this interval as possible. It is this interval that determines the frequency resolution of our DFT.

$$\text{BinWidth} = \frac{1}{T_s} \quad \text{where } T_s = \text{Duration of the signal.}$$

Note that the digitizer sampling frequency has no effect on frequency resolution in this regard. The sampling frequency still impacts the measurement with respect to over-sampling and under-sampling issues which are not taken up in this paper.

A Decimation-In-Time or Decimation-In-Frequency Fast Fourier Transform algorithm can be used to compute all N points of the DFT. This requires $\left(\frac{N}{2}\right) \log_2 N$ complex multiplications and $\left(\frac{N}{2}\right) \log_2 N$ complex additions. It is also required that N be a power of two to implement the FFT.

The Goertzel algorithm's processing requirements are lower. This algorithm computes the results of an N point Discrete Fourier Transform, DFT, for a single frequency point, k . This algorithm requires $2 \cdot (N+2)$ real multiplications and $4 \cdot (N+1)$ real additions. Another benefit is that N is not required to be a power of two.

Therefore, given a fixed interval in which to digitize a signal, N can be chosen to more closely maintain the highest frequency resolution of the result.

II.2 Computation

Computation of the Goertzel algorithm can be divided into two phases. The first phase involves computing the feedback legs in Figure II.2.1 and performing this iterative equation N times using the N input values of $x(n)$. The second phase evaluates $X(k)$ by computing the feedforward leg in Figure II.2.2.

II.2.1 The Feedback Phase

The feedback phase occurs for N input samples (counted as $n=0, 1, \dots, N-1$). During this phase, two intermediate values $Q(n-1)$ and $Q(n-2)$ are stored in data memory. Their values are evaluated iteratively as follows:

$$Q(n) = C_1 \cdot Q(n-1) - Q(n-2) + x(n)$$

$$Q(n-2) = Q(n-1)$$

$$Q(n-1) = Q(n)$$

Where:

$$C_1 = 2 \cdot \cos(2\pi k / N)$$

$$Q(-2) = Q(-1) = 0$$

$$n = 0, 1, 2, \dots, N-1$$

$$k = DFT \text{ frequency index of interest}$$

$$N = \text{number of samples used}$$

$$x(n) = \text{current input sample}$$

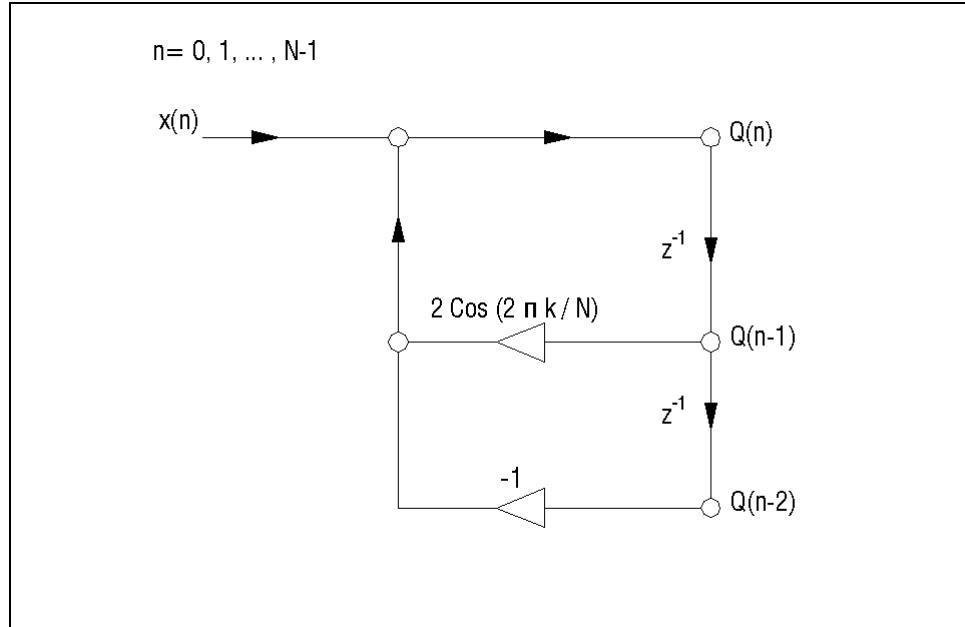


Figure II.2.1 The feedback phase of the Goertzel algorithm.

II.2.2 The Feedforward Phase

The feedforward phase occurs once after the feedback phase has been performed for N input samples. The feedforward phase generates the complex DFT point value $X(k)$.

$$X(k) = Q(N-1) - Q(N-2) \cdot C_2 + j Q(N-2) \cdot C_3$$

Where:

$$C_2 = \cos(2\pi k / N)$$

$$C_3 = \sin(2\pi k / N)$$

Note that the coefficients C_1 , C_2 , and C_3 are constants given k and N and can be computed beforehand.

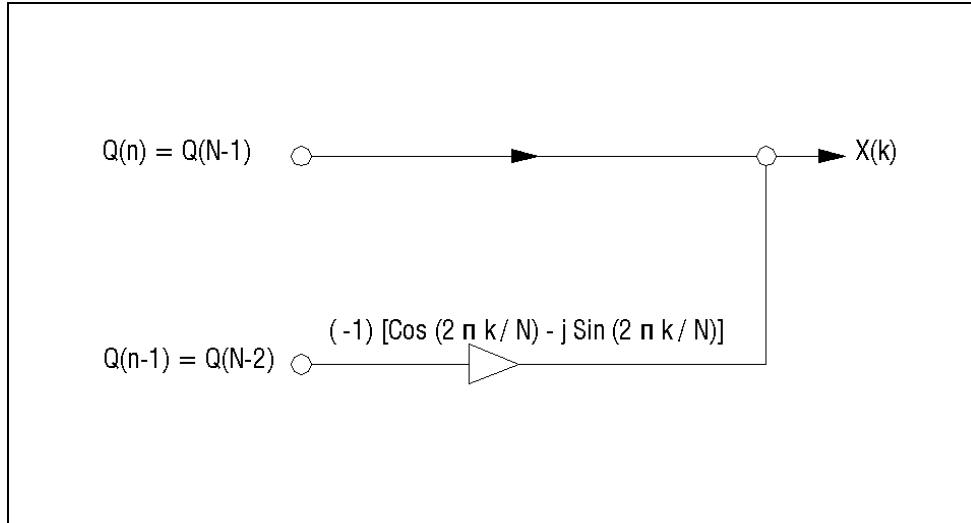


Figure II.2.2 The feedforward phase of the Goertzel algorithm.

II.3 Optimizing the Computation

The Goertzel algorithm has the same frequency characteristics as the DFT algorithm. Particularly, when an N -point DFT is performed on a data sequence sampled at frequency F_s , the output frequency samples, or frequency bins, are equally spaced $\frac{F_s}{N}$ apart. In order to ensure that all of the energy at a particular frequency F_0 is captured in a single frequency bin we need to meet the following criteria as nearly as possible:

$$F_0 = k \cdot \frac{F_s}{N} \quad \text{where } k \text{ is an integer.}$$

or

$$T_s = k \cdot \frac{1}{F_0} \quad \text{where } T_{ss} = \text{Duration of the signal.}$$

The effects of “leakage” out of a particular frequency bin is described using time and frequency domain plots in [4]. Since the Goertzel algorithm does not require that N be a power of two, N and F_s can be adjusted to meet the requirements above while minimally reducing the signal interval used. Recall that the longer the signal interval represented by the N samples taken at the rate of F_s , the higher the frequency resolution of the result.

All of this implies a procedure for determining the parameters, k , T_s , F_s , and N .

1. Certainly, the target frequency of interest is fixed at F_0 .
2. Calculate a preliminary value for k : $\bar{k} = F_0 \cdot T_S^{\max}$ where T_S^{\max} is the maximum interval in which we have a signal or some other limit on the signal interval over which we may digitize the signal.
3. Compute the actual signal interval that will be represented by the digitized data: $k = \text{IntegerPart}[\bar{k}] + 1$ and $T_S = \frac{k - 1}{F_0}$
4. The digitizer will have a maximum sampling frequency F_S^{\max} for which we can compute a preliminary value for N : $\bar{N} = F_S^{\max} \cdot T_S$
5. Compute the desired sampling frequency and N : $N = \text{IntegerPart}[\bar{N}]$ and $F_S = \frac{N}{T_S}$.

II.2 Simulation

Using Mathematica or MatLab we can explore the operation of the algorithm and get an idea of how it would perform in a particular application. The readout of a Beam Position Monitor was simulated and the results are shown below. The code that implements the simulation of the BPM readout is given in the appendices. Two Trials are shown below. Trial two shows the results with varying levels of random noise added to the signal.

After writing my own implementation I discovered that Mathworks has already implemented the algorithm in its signal processing toolbox.

<http://www.mathworks.com/access/helpdesk/help/toolbox/signal/goertzel.shtml>

Trial 1 (Test1)

Target Frequency of Interest,	F_0	= 53.1047 MHz.
Maximum Period Signal is Present,	T_S^{\max}	= 1.6 microseconds.
Maximum Sampling Frequency,	F_S^{\max}	= 64 MHz.
DFT Target Frequency Index,	k	= 85.
Number of Samples Used,	N	= 101.
Actual Signal Interval,	T_S	= 1.582 microseconds.
Optimal Sampling Frequency,	F_s	= 63.852 MHz.
Beam Intensity,	Int	= 3 e10 ppb. (Vp = 2 Volts)
ADC Bit Resolution,	Bits	= 12.
ADC Input Range,	Range	= 5.0 Volts

Simulation Results (values in mm)

Expected	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5
Computed	0.498	0.998	1.496	1.998	2.494	2.995	3.495	3.995	4.492	4.995	5.492
Delta	0.002	0.002	0.004	0.002	0.006	0.005	0.005	0.005	0.008	0.005	0.008

Overall RMS error = 0.0048 mm

Trial 2 (Test2)

Target Frequency of Interest,	F_0	= 53.1047 MHz.
Maximum Period Signal is Present,	T_S^{\max}	= 1.6 microseconds.
Maximum Sampling Frequency,	F_S^{\max}	= 64 MHz.
DFT Target Frequency Index,	k	= 85.
Number of Samples Used,	N	= 101.
Actual Signal Interval,	T_S	= 1.582 microseconds.
Optimal Sampling Frequency,	F_s	= 63.852 MHz.
Beam Intensity,	Int	= 3 e10 ppb. ($V_p = 2$ Volts)
ADC Bit Resolution,	Bits	= 12.
ADC Input Range,	Range	= 5.0 Volts
Added Noise Voltage		Uniformly dist. volt. in the range +/- Vnoise

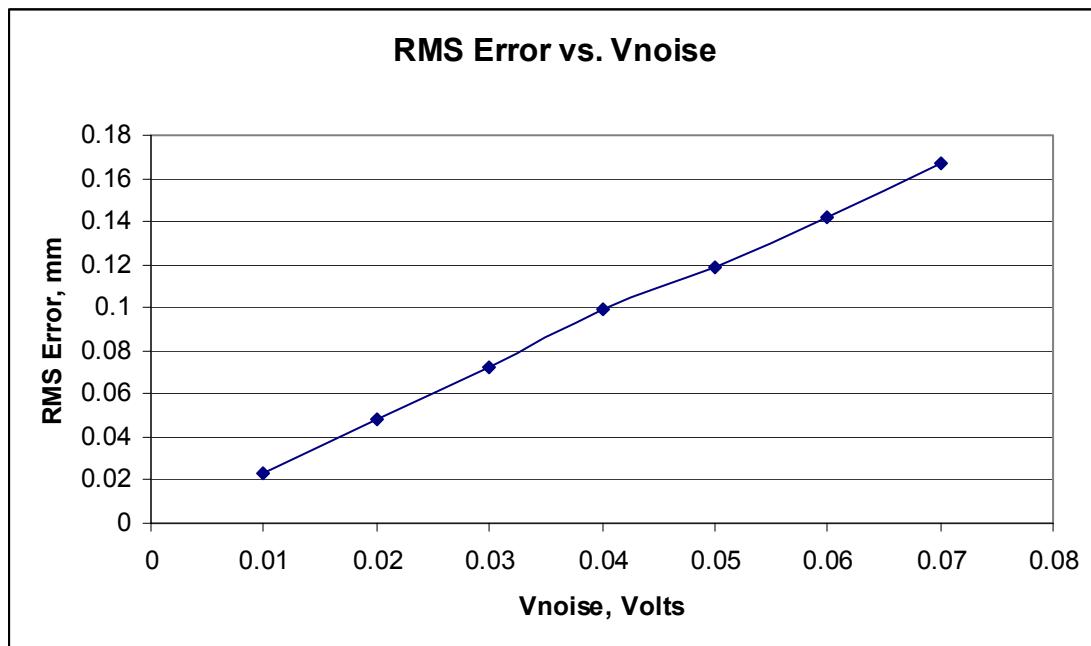


Figure 1 Plot of RMS error in Trial 2.

References:

- [1] Banks, Kevin, “The Goertzel Algorithm”, Embedded Systems Programming, August 8, 2002.
- [2] Analog Devices, “Digital Signal Processing Applications”, Vol.1, Chap 14.
- [3] Oppenheim, A. V., Schafer, R. W., “Discrete-Time Signal Processing”, Englewood Cliffs, N.J. : Prentice Hall, c1989, ISBN: 013216292X
- [2] Analog Devices, “Digital Signal Processing Applications”, Vol.1, Chap 6.7.

APPENDIX A

Mathematica Code for Goertzel Simulation

```

BeginPackage["GoertzelFFT`"]

TestVector::usage = "TestVector[BeamInt_, PosStep_, Len_] generates a 2
by [Len] array of voltages that correspond to the output of a standard
Main Injector style BPM given a specific beam intensity [BeamInt] in
1x10^10 ppb. Computation of the voltages is done according to the
equations developed in [NUMI BPM Detectors], V. Makeev, 2002, FNAL.
Each pair of voltages in the array will represent a sequence of beam
position steps. The number of steps are determined by the variable
[Len] and the size of each step by [PosStep]. The final array has the
form {{chA1,chB1},{chA2, chB2}, ... }."

GoertzelFFT1::usage = "GoertzelFFT1[x_,k_] computes a single point FFT.
Varaible [x] is the list of sampled time data and [k] is the index of
the frequency point in the DFT you wish to determine."

GoertzelFFT2::usage = "GoertzelFFT2[x_,k_] performs the same function
as GoertzelFFT1, but in the more efficient manner. See

[1] Analog Devices, [Digital Signal Processing Applications], Vol.1,
Chap 14.
[2] Oppenheim, A. V., Schafer, R. W., [Discrete-Time Signal
Processing], Englewood Cliffs, N.J. : Prentice Hall, c1989, ISBN:
013216292X "

GenerateBurst::usage = "GenerateBurst[VAmplitude_, Fo_, Fs_, Ts_]
generates 2 sinewave signals of frequency, [Fo] (Hz), and zero-to-peak
amplitude given by the pair [VAmplitude] { chA Vpk, chB Vpk }, sampled
at the rate [Fs] (Hz) for the given time interval [Ts] (sec)."

GenerateBurstPlusNoise::usage =
"GenerateBurstPlusNoise[VAmplitude_, Vnoise_, Fo_, Fs_, Ts_] performs
the same function as GenerateBurst except that random noise is added
according to Random[Real,{-Vnoise,Vnoise}]."

GenerateBurstPlusNoise2::usage = " "

QuantizeData::usage = "QuantizeData[data_, bits_, range_, offset_]
quintiles the amplitude of the numbers in [data], as if sampled by an
ADC with bit resolution, [bits], and a full scale voltage of, [range]
and offset, [offset]."

Test0::usage = "Test0[Vp_, Fs_, Ts_, Fo_, Kfo_] examines the variance
of the results of the Goertzel computation with respect to
the phase of the input signal."

Test1::usage = "Test1[BeamInt_, Fs_, Ts_, Fo_, Kfo_] simulates the BPM
signals for a number of beam positions and then compares these expected

```

positions to those computed using the Goertzel algorithm. The function outputs the test vector of voltages used in the simulation"

```

Test2::usage = "Test2[BeamInt_, Vnoise_, Fs_, Ts_, Fo_, Kfo_] test with
added noise"

Test3::usage = "Test3[BeamInt_, Vnoise_, Fs_, Ts_, Fo_, Kfo_] "

Begin["`Private`"]
(*=====
TestVector[BeamInt_, PosStep_, Len_]:=Module[{VV, pos, i, Kg, Kc, sigma, L0, delta, b},
(* Parameters for the Standard Transport BPM
   Taken from paper "NUMI BPM Detectors", V. Makeev, FNAL*)
= 0.774;
= 0.907;
= 2*10^-9 (*seconds* );
= 98.298 (* mm * );
= 0.4 (* mm * );
= 49.149 (* mm * );

(* Numi BPM output signal amplitude, Volts-peak/1x10^10 ppb *)
[x_] = 9.362*10^-3 Exp[(-0.055)(sigma^2)] L0(1 + Kg Kc(x+delta/2)/
b)/(2 Kg);
[x_] = 9.362*10^-3 Exp[(-0.055)(sigma^2)] L0(1 - Kg Kc(x-delta/2)/
b)/(2 Kg);

= {{Ur[0]*BeamInt,Ul[0]*BeamInt}};
= 0;
[ i = 1, i <= Len, i++,
  pos = pos + PosStep;
  AppendTo[ VV, {Ur[pos]*BeamInt,Ul[pos]*BeamInt} ];
];
VV
]

(*=====
GoertzelFFT1[x_, k_] :=Module[{i, n, y, Len, Wnk, yy},
Len = Length[x]; (* This is the slower direct version *)
y = Array[0, Len];
y[[1]] = x[[1]];
Wnk = Evaluate[Cos[2 Pi k / Len]] + I Evaluate[Sin[2 Pi k / Len]];
For[ i = 2, i <= Len, i++,
  y[[i]] = Evaluate[ x[[i]] + y[[i - 1]] * Wnk]
];
yy = y[[Len]];
{Abs[yy], Arg[yy]}
]

(*=====
GoertzelFFT2[x_, k_] :=Module[{i, Q, C, y, Len, Wnk},
Len = Length[x];
(*Pre Computed Coefficients*)
C = {2*Evaluate[Cos[2 Pi k / Len]],
Evaluate[Cos[2 Pi k / Len]],

```

```

        Evaluate[Sin[2 Pi k / Len]]
    };
(*Initial Condition*)
Q = Table[0, {3}];
(*N Iterations -- Feedback Phase*)
For[ i = 1, i <= Len, i++,
    Q[[1]] = C[[1]]*Q[[2]] - Q[[3]] + x[[i]];
    Q[[3]] = Q[[2]];
    Q[[2]] = Q[[1]];
];
(*Feedforward phase, y = X(k) the kth frequency point of the
DFT[x(n)]*)
y = Q[[2]] - C[[2]]*Q[[3]] + I C[[3]]*Q[[3]];
Sqrt[Re[y]^2 + Im[y]^2]
]

(*=====
GenerateBurst[VAmplitude_, Fo_, Fs_, Ts_] :=
Module[{ VV, VA, VB, VchA, VchB},
VchA = VAmplitude[[1]];
VchB = VAmplitude[[2]];
VA = Table[VchA Sin[2 Pi Fo t ]+2.5,{t,0,Ts, 1/(Fs)}];
VB = Table[VchB Sin[2 Pi Fo t ]+2.5,{t,0,Ts, 1/(Fs)}];
VV = {VA,VB}
]

(*=====
GenerateBurstPlusNoise[VAmplitude_, Vnoise_, Fo_, Fs_, Ts_] :=
Module[{ VV, VA, VB, VchA, VchB},
VchA = VAmplitude[[1]];
VchB = VAmplitude[[2]];
VA = Table[Random[Real,{-Vnoise,Vnoise}]
          + VchA Sin[2 Pi Fo t ],{t, 0, Ts, 1/(Fs)}];
VB = Table[Random[Real,{-Vnoise,Vnoise}]
          + VchB Sin[2 Pi Fo t ],{t, 0, Ts, 1/(Fs)}];
VV = {VA,VB}
]

(*=====
GenerateBurstPlusNoise2[VAmplitude_, Vnoise_, Fo_, Fs_, Ts_] :=
Module[{ VV, VA, VB, VchA, VchB},
VchA = VAmplitude[[1]];
VchB = VAmplitude[[2]];
VA = Table[
    Random[Real,{-Vnoise,Vnoise}] Sin[2 Pi (Fo + 2*^6) t + Random[]
Pi]
    + Random[Real,{-Vnoise,Vnoise}] Sin[2 Pi (Fo - 2*^6) t + Random[]
Pi]
    + VchA Sin[2 Pi Fo t ],{t, 0, Ts, 1/(Fs)}
];
VB = Table[
    Random[Real,{-Vnoise,Vnoise}] Sin[2 Pi (Fo + 2*^6) t + Random[]
Pi]
    + Random[Real,{-Vnoise,Vnoise}] Sin[2 Pi (Fo - 2*^6) t + Random[]
Pi]
    + VchB Sin[2 Pi Fo t ],{t, 0, Ts, 1/(Fs)}
];

```

```

VV = {VA,VB}
]

(*=====
QuantizeData[data_, bits_, range_, offset_] :=
  N[IntegerPart[(data + offset) * (2^bits/range)]]

(*=====
<<Statistics`DescriptiveStatistics`; (* include this package *)

(*=====
Test0[Vp_, Fs_, Ts_, Fo_, Kfo_] :=
Module[{i, sig, base, sig2, b, ki,
       Koutm = List[],
       Koutp = List[],
       ADCBitRes = 12,      (*assumed ADC bit resolution*)
       ADCVoltRng = 5,      (*assumed ADC input voltage range*)
       ADCZeroOffset = 0    (*assumed ADC zero voltage offset*)
},
For[i=1,i<=20,i++,
  b= Pi i/10;
  sig = QuantizeData[
    Table[Vp Cos[2 Pi Fo t + b ] + 2.5, {t,0,Ts, 1/(Fs)}],
    ADCBitRes, ADCVoltRng, ADCZeroOffset];
  ki = GoertzelFFT1[sig , Kfo];
  AppendTo[Koutm, ki[[1]] ];
  AppendTo[Koutp, ki[[2]] ];
];
(* Output of Module =====)
{Koutm, Koutp, Fs*Ts}
]

(*=====
Test1[BeamInt_, Fs_, Ts_, Fo_, Kfo_] :=
Module[{i,TV, Len, sig1, sig2, chA, chB,
       chanAB = {{0,0}},      (*record of channel A & B magnitudes*)
       pos = {0},            (*position computed using Goertzel*)
       possx = {0},           (*expected position as given*)
       error = {0},           (*error between expected and computed*)
       ADCBitRes = 12,        (*assumed ADC bit resolution*)
       ADCVoltRng = 5,        (*assumed ADC input voltage range*)
       ADCZeroOffset = 0,      (*assumed ADC zero voltage offset*)
       Sx = 70.113,           (*assumed BPM detector sensitivity*)
       PosStep = 0.5,          (*given position step for test*)
       LastPos = 6             (*last position in test*)
},
Len = LastPos/PosStep;
TV = TestVector[BeamInt,PosStep, Len];

For[i=2, i<= Len, i++,
  sig1 = GenerateBurst[ TV[[i]], Fo, Fs, Ts];
  sig2 = QuantizeData[ sig1, ADCBitRes, ADCVoltRng,
ADCZeroOffset];
  chA = GoertzelFFT2[ sig2[[1]] , Kfo];
  chB = GoertzelFFT2[ sig2[[2]] , Kfo];
]

```

```

AppendTo[chanAB, {chA, chB} ];
AppendTo[pos, Sx*(chA-chB)/(chA+chB) ];
AppendTo[posx, posx[[i-1]] + PosStep ];
AppendTo[error, pos[[i]] - posx[[i]] ];
];
(* Output of Module =====)
{TV, chanAB, posx, pos, N[ Sqrt[ Mean[error^2] ] ] }
]

(*=====
Test2[BeamInt_, Vnoise_, Fs_, Ts_, Fo_, Kfo_] :=
Module[{i, TV, Len, sig, chA, chB,
(* Vnoise, +/- random noise voltage setting*)
pos = Table[0,{i,13}], (*position computed using Goertzel*)
posx = Table[0,{i,13}], (*expected position as given*)
error = {0}, (*error between expected and computed*)
ADCBitRes =12, (*assumed ADC bit resolution*)
ADCVoltRng =5, (*assumed ADC input voltage range*)
ADCZeroOffset =0, (*assumed ADC zero voltage offset*)
Sx = 70.113, (*assumed BPM detector sensitivity*)
PosStep = 0.5, (*given position step for test*)
LastPos = 6 (*last position in test*)
},
Len = LastPos/PosStep;
TV = TestVector[BeamInt, PosStep, Len];
For[i=2, i<= Len, i++,
For[j=0, j<=40, j++,
sig = GenerateBurstPlusNoise[ TV[[i]], Vnoise, Fo, Fs,
Ts];
sig = QuantizeData[ sig, ADCBitRes, ADCVoltRng,
ADCZeroOffset];
chA = GoertzelFFT2[ sig[[1]], Kfo];
chB = GoertzelFFT2[ sig[[2]], Kfo];
pos[[i]] = Sx*(chA-chB)/(chA+chB);
posx[[i]] = posx[[i-1]] + PosStep;
AppendTo[error, pos[[i]] - posx[[i]] ];
]
];
(* Output of Module =====)
{posx, pos, N[ Sqrt[ Mean[error^2] ] ] }
]

(*=====
Test3[BeamInt_, Vnoise_, Fs_, Ts_, Fo_, Kfo_] :=
Module[{i, j, TV, Len, sig, chA, chB,
(* Vnoise, +/- random noise voltage setting*)
pos = Table[0,{i,13}], (*position computed using Goertzel*)
posx = Table[0,{i,13}], (*expected position as given*)
error = {0}, (*error between expected and computed*)
ADCBitRes =12, (*assumed ADC bit resolution*)
ADCVoltRng =5, (*assumed ADC input voltage range*)
ADCZeroOffset =0, (*assumed ADC zero voltage offset*)
Sx = 70.113, (*assumed BPM detector sensitivity*)
PosStep = 0.5, (*given position step for test*)

```

```
LastPos = 6           (*last position in test*)
},
Len = LastPos/PosStep;

TV = TestVector[BeamInt,PosStep, Len];

For[i=2, i<= Len, i++,
  For[j=0, j<=40, j++,
    sig = GenerateBurstPlusNoise2[ TV[[i]], Vnoise, Fo, Fs,
Ts];
    sig = QuantizeData[ sig, ADCBitRes, ADCVoltRng,
ADCZeroOffset];
    chA = GoertzelFFT2[ sig[[1]] , Kfo];
    chB = GoertzelFFT2[ sig[[2]] , Kfo];
    pos[[i]] = Sx*(chA-chB)/(chA+chB);
    posx[[i]] = posx[[i-1]] + PosStep;
    AppendTo[error, pos[[i]] - posx[[i]] ];
  ]
];
(* Output of Module =====)
{posx,pos, N[ Sqrt[ Mean[error^2] ] ] }
]

End[]
(*Protect[GoertzelFFT]*)
EndPackage[]
```

APPENDIX B

MatLab Code for Goertzel Simulation

```

function [posx, pos, rmserr] = Test1(Beamint, Fs, Ts, Fo, Kfo)
% This is the first of the Goertzel Tests
%
% THE COMMAND LINE USED FOR THE REPORT WAS:
% [px,p,rms]=Test1(3.5, 63.852e6, 1.582e-6, 53.1047e6, 85);

    pos = [0];           %(*position computed using Goertzel*)
    posx = [0];          %(*expected position as given*)
    err = [0];           %(*error between expected and computed*)
    ADCBitRes =12;       %(*assumed ADC bit resolution*)
    ADCVoltRng =5;       %(*assumed ADC input voltage range*)
    ADCZeroOffset = 0;   %(*assumed ADC zero voltage offset*)
    Sx = 70.113;         %(*assumed BPM detector sensitivity*)
    PosStep = 0.5;        %(*given position step for test*)
    LastPos = 6;          %(*last position in test*)
    Vnoise = 0.0;

    Len = LastPos / PosStep;
    TV = TestVector(Beamint, PosStep, Len);

    for i=2:Len
        [s1A,s1B] = GenerateBurstPlusNoise( TV(i,:), Vnoise, Fo, Fs,
        Ts);
        sA = QuantizeData( s1A, ADCBitRes, ADCVoltRng, ADCZeroOffset);
        sB = QuantizeData( s1B, ADCBitRes, ADCVoltRng, ADCZeroOffset);
        [mA,pA] = GoertzelFFT2(sA , Kfo);
        [mB,pB] = GoertzelFFT2(sB , Kfo);

        pos = [pos , Sx*(mA-mB) / (mA+mB) ];
        posx = [posx, posx(i-1) + PosStep ];
        err = [err , pos(i) - posx(i) ];

    end

    rmserr = sqrt(err*err');

    %=====
function y = QuantizeData(data, bits, range, offset)
% Converts the data into quantized digital words

y = floor( (data + offset) * (2^bits/range) );

    %=====
function [mag, phase] = GoertzelFFT2(x, k)
% The Goertzel single point DFT algorithm

Len = length(x);

```

```

% (*Pre Computed Coefficients*)
C = [2*cos(2*pi*k / Len), cos(2*pi*k / Len), sin(2*pi*k / Len)];
A = [C(1), -1; 1, 0];
B = [1; 0];
H = [1, -C(2)+j*C(3)];

% (*Initial Condition*)
Q = [0; 0];

% (*N Iterations -- Feedback Phase*)
for i = 1:Len
    Q = A*Q + B*x(i);
end

% (*Feedforward phase, y = X(k)
%   the kth frequency point of the DFT[x(n)]*)
y = H*Q;

mag = abs(y)/Len;
phase = angle(y);

%(*=====
function V = TestVector(BeamInt, PosStep, Len)
% Produces signal peak voltages for BPM channel A and B
% for a sequence of beam positions given a beam intensity.

    pos = 0;
    [Ur, Ul] = bpmvolts(pos);
    V = [Ur*BeamInt, Ul*BeamInt];
    for i = 1:Len
        pos = pos + PosStep;
        [Ur, Ul] = bpmvolts(pos);
        V = [V; Ur*BeamInt, Ul*BeamInt];
    end

%(*=====
function [Ur, Ul] = bpmvolts(pos)
% Produces a pair of per 1e10 ppb intensity, bpm voltages.

%(* Parameters for the Standard Transport BPM
%   Taken from paper "NUMI BPM Detectors", V. Makeev, FNAL*)

    Kg = 0.774;
    Kc = 0.907;
    sigma = 2e-9; %(*seconds*);
    L0 = 98.298; %(* mm *);
    delta = 0.4; %(* mm *);
    b = 49.149; %(* mm *);

    % (* Numi BPM output signal amplitude, Volts-peak/1x10^10 ppb *)
    Ur = 9.362e-3*exp((-0.055)*(sigma^2))*L0*(1 + Kg*Kc*(pos+delta/2) /
b)/(2*Kg);
    Ul = 9.362e-3*exp((-0.055)*(sigma^2))*L0*(1 - Kg*Kc*(pos-delta/2) /
b)/(2*Kg);

%(*=====

```

```
function [VA,VB] = GenerateBurstPlusNoise(VAmplitude, Vnoise, Fo, Fs, Ts)
%
VchA = VAmplitude(1);
VchB = VAmplitude(2);
t=[0:(1/Fs):Ts];

VA = (rand-0.5)*2*Vnoise + VchA*sin(2*pi*Fo*t);
VB = (rand-0.5)*2*Vnoise + VchB*sin(2*pi*Fo*t);
```